TDDD86 – Laboration #8

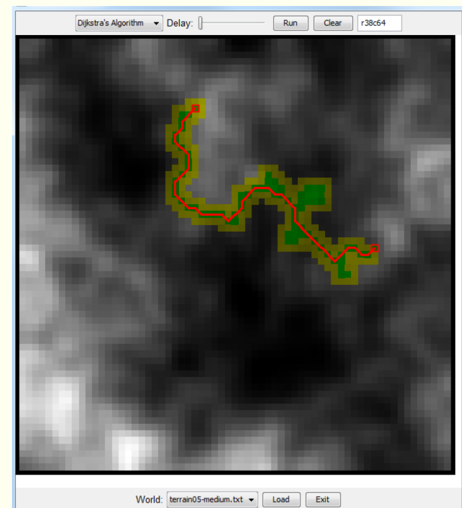
6 september 2019

I den här uppgiften får du fokusera på grafer, speciellt att söka efter vägar i grafer. Filerna du behöver för att komma igång finns som `labb8.tar.gz` på kurshemsidan.

Redovisning: Efter att du redovisat muntligt, gör en `git commit -m ''TDDD86 Lab 8 redovisning''` och en `git push`. Skicka sedan ett mail till din assistent med ämnet: `[TDDD86] Lab 8 redovisning` . Se till att filen **trailblazer.cpp** är med.

Trailblazer

Det givna programmet är tänkt att visa olika 2D-världar som representerar antingen labyrinter eller terräng och låter användaren generera vägar i världen från en punkt till en annan. När du startar programmet ser du en 2D-labyrint där vita kvadrater är öppna och svarta representerar väggar. Programmet kan också visa slumpgenererad terräng där ljusa färger indikerar högre elevation och mörkare färger lägre elevation. Bergskedjor är ljusa, medan djupa dalar är närmast svarta.



Om du klickar på två punkter i världen hittar programmet en väg från startpunkt till slutpunkt. Under tiden programmet gör det färglägger det noderna gröna, gula och gråa baserat på färgerna de tilldelas av algoritmen. När vägen väl är funnen markeras vägen och information om den skrivs ut till konsollen. Användaren kan välja en av fyra vägsökningsalgoritmer i översta menyn:

- djupetförstsökning (DFS)
- breddenförstsökning (BFS)
- Dijkstras algoritm
- A*-sökning

Fönstret innehåller flera kontrolldon. Du kan ladda labyrinter och terränger av olika storlek från menyn längst ned och sedan klicka på "Load"-knappen.

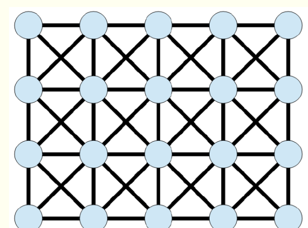
Funktioner som måste implementeras

I din **trailblazer.cpp**-fil måste du implementera följande fyra funktioner för att söka efter vägar i grafer:

```
vector<Node*> depthFirstSearch(BasicGraph& graph, Node* start, Node* end);
vector<Node*> breadthFirstSearch(BasicGraph& graph, Node* start, Node* end);
vector<Node*> dijkstrasAlgorithm(BasicGraph& graph, Node* start, Node* end);
vector<Node*> aStar(BasicGraph& graph, Node* start, Node* end);
```

Varje funktion implementerar en av fyra grafsökningsalgoritmer vi diskuterat i kursen. Du ska söka i den givna grafen efter en väg från den givna startnoden till den givna slutnoden. Om du hittar en sådan väg, ska vägen du returnerar vara en uppräkning av alla noder längs den vägen med startnoden först (index 0 i vektorn) och slutnoden sist. Om ingen sådan väg finns, returnera en tom vektor. Du får anta att grafen som passeras till funktionen har ett giltigt tillstånd och start- och slutnoderna är giltiga icke-NULL-noder i den givna grafen.

Det tillhandahållna huvudprogrammet låter dig testa varje algoritm för sig innan du går vidare till nästa. Du får lägga till fler funktioner som hjälpare om du vill, särskilt för att hjälpa dig att implementera eventuellt rekursiva algoritmer och/eller minska redundans mellan olika algoritmer innehållande liknande kod.



Den tvådimensionella världen representeras av en `BasicGraph`, där varje nod representerar en specifik plats i världen. Om det är en labyrint representerar varje position en kvadrat i labyrintens rutnätsliknande värld. Öppna kvadrater är sammanbundna med bågar till andra närliggande öppna kvadrater som ligger i direkt

anslutning till dem (skiljer sig med exakt +/- 1 rad eller kolumn). Svarta "vägg"-kvadrater är inte sammanbundna som grannar till några andra kvadrater. Om världen är en terräng snarare än en labyrint representerar varje position en elevation mellan 0 (lågland) och 1 (hög bergstopp). Terrängpositioner är sammanbundna i samtliga 8 riktningar inklusive diagonala grannar, medan labyrintpositioner endast är sammanbundna direkt upp, ner, vänster och höger.

Din kod kan behandla labyrinter och terräng på exakt samma sätt. Du behöver bara tänka på de olika världstyperna som grafer med noder och bågar som sammanbinder närliggande noder. Dina vägsökningsalgoritmer kommer att fungera oavsett vilken graf de anropas på.

Tillhandahållen kod

Den här uppgiften har mycket tillhandahållen kod. Nedan följer en liten genomgång av vad varje fil innehåller. Det är inte nödvändigt att undersöka eller känna till alla filer för att kunna slutföra uppgiften.

- **trailblazer.h / .cpp** Vi tillhandahåller skelettversioner av dessa filer där du ska skriva vägsökningskoden för den här uppgiften. Du ska inte ändra några andra filer än **trailblazer.cpp**
- **adapter.h / .cpp** Fungerar som en brygga mellan GUI:t och dina grafalgoritmer.
- **BasicGraph.h / .cpp** Definierar `BasicGraph`-datastrukturen som representerar grafen för labyrinten eller terrängen. samt relaterade typer som `Node` och `Edge`.
- **costs.h / .cpp** Definierar funktioner för att beräkna kostnaden för att flytta från en kvadrat till en annan i terränger.
- **trailblazergui.h / .cpp** Applikationens GUI och `main`-funktion.
- **types.h / .cpp** Definierar vissa understödjande typer använda av annan tillhandahållen kod i uppgiften.

En graf består av noder och bågar. Varje nod representeras av en instans av typen `Node`, som också har aliaset `Vertex`. Varje `Node` har följande medlemmar:

`Node (Vertex)` member	Description
`string name`	vertex's name, such as `"r34c25"` or `"vertex17"`
`Set<Arc*> arcs`	edges outbound from this vertex
`double cost`	cost to reach this vertex (initially `0`)
`bool visited`	whether this vertex has been visited yet (initially `false`)
`Node* previous`	pointer to a vertex that comes before this one; initially `NULL`
`void setColor(Color c)`	sets this vertex to be drawn in the given color in the GUI, one of `WHITE`, `GRAY`, `YELLOW`, or `GREEN`
`Color getColor()`	returns color you set previously using `setColor`; initially `WHITE`
`double heuristic(Node* other)`	returns an admissible heuristic cost estimate for traveling from this vertex to the given other vertex *(used by the A* algorithm; described later)*
`void resetData()`	sets `cost`, `visited`, and `previous` back to their initial values
`string toString()`	returns a printable string representation of the vertex for debugging

Medlemsvariablerna du är tänkt att använda i dina sökalgoritmer är `cost`, `visited` och `previous`. Flera av algoritmerna behöver "märka" noder som besökta eller associera en aktuell kostnad med en nod eller hålla reda på pekare från en nod till en annan för att kunna rekonstruera en väg. Använd dessa medlemmar i varje nod för att hålla reda på sådan information. Du kan anropa `resetData` på en nod för att radera dessa data eller på en `BasicGraph` som helhet för att radera allt sådant data för alla noder.

Varje båge i grafen representeras av en instans av `Edge`-strukturen, som också har aliaset `Arc`. Varje `Edge` har följande medlemmar:

`Arc (Edge)` member	Description
`Node* start`	the starting vertex of this edge
`Node* finish`	the ending vertex of this edge (i.e., `finish` is a neighbor of `start`)
`double cost`	cost to traverse this edge
`string toString()`	returns a printable string representation of the `edge` for debugging

Noder och bågar skickas till dina algoritmfunktioner som en del av ett `BasicGraph`-objekt. Klassen `BasicGraph`

är en variant av Stanfords `Graph`-klass; tekniskt sett är den en subklass till `Graph<Node, Arc>`. Via arv exponerar `BasicGraph` en stor mängd metoder för alla upptänkliga saker man kan vilja göra med en graf. Du behöver dock bara bekymra dig om att kunna nollställa grafen i början av varje anrop till en vägsökningsalgoritm (med `resetData()`) samt att kunna iterera över noder och bågar på olika sätt. Följande kod skriver ut alla noder och bågar i grafen `graph`. Därefter skrivs alla bågar som startar i noden `start` ut, följt av alla grannar till noden `start`.

```
for (Node *node : graph.getNodeSet())
    cout << *node << endl;
for (Edge *edge : graph.getEdgeSet())
    cout << *edge << endl;
for (Edge *edge : graph.getEdgeSet(start))
    cout << *edge << endl;
for (Node *node : graph.getNeighbors(start))
    cout << *node << endl;
```

Implementationsdetaljer

Förutom att söka efter en väg i varje algoritm vill vi att du lägger till kod för att lägga till färg till olika noder vid vissa tillfällen. Färginformationen används av GUI:t för att visa hur din algoritm fortskrider. För att färglägga en nod, anropa `color`-medlemsfunktionen i nodens `Node`-objekt med en global färgkonstant som argument.

```
Node* myNode = graph.getNode("foo");
myNode->color(GREEN); // set the node's color to green
```

Nedan följer en lista med färger och när du ska använda dem:

- köad = gul: När du köar en nod för att besökas för första gången, som i BFS eller Dijkstra när du sätter in en nod i datastrukturen för senare behandling, färga den gul (`YELLOW`).
- besökt/behandlad = grön: När din algoritm faktiskt besöker och behandlar en viss nod, som när den tas ut ur kön i BFS eller Dijkstra, eller när den är startnoden för ett rekursivt anrop till DFS, färga den grön (`GREEN`).
- eliminerad = grå: När din algoritm har behandlat färdigt en nod och inte hittat en väg från den noden och därför "ger upp" på den noden som kandidat, färga den grå (`GRAY`). Den enda algoritmen som gör backtracking på det här sättet är DFS.

Det inkluderade GUI:t har en slider som du kan dra för att sätta en fördröjning mellan färgläggningsanrop. Defaultpositionen orsakar ingen fördröjning alls men om du drar slidern längst till höger får varje anrop till `color` GUI:t att stanna upp en kort stund så att du kan se din algoritm utföra sitt arbete.

DFS: Algoritmen kan implementeras rekursivt eller iterativt, valet är upp till dig. En rekursiv lösning kan ibland bli långsam eller krascha på stora världar; detta är ok. Du behöver inte modifiera din DFS-implementation för att undvika krascher som beror på för stor anropsstackstorlek.

BFS: Din kod behöver återskapa stigen den hittar, så titta på pseudokoden för Dijkstras algoritm för inspiration. En intressant observation är att BFS och Dijkstra beter sig exakt likadant på labyrinter men olika på terränger. (Varför?)

Dijkstras algoritm: Både Dijkstras algoritm och A* använder sig av en prioritetskö av noder att behandla och kräver dessutom att prioritetskön kan ändra en given nods prioritet i kön — något som `std::priority_queue` inte klarar av. Du ska därför använda dig av Stanfords klass `pqueue.h` som har medlemmen `changePriority`.

`PriorityQueue` member	Description
`changePriority(value, newPriority)`	updates the given element in the priority queue to use the given new priority, rearranging the internal heap ordering to maintain consistency; throws an error if the value is not found or if the new priority is less urgent (greater) than the old existing priority for that value

Observera att en given nods aktuella prioritet kan lagras på två platser i din kod: i kostnadsfältet `cost` i själva `Node`-strukturen och i prioritetsköns ordning. Du måste själv se till att uppdatera dessa och om värdena inte hålls synkroniserade kan det orsaka buggar i ditt program.

A*: Algoritmen för A*-sökning är, i allt väsentligt, en variant av Dijkstras algoritm som använder en heuristik för att trimma ordningen av element i sin prioritetskö så att troligtvis mer gynnsamma element behandlas först. Detta betyder att du behöver en heuristikfunktion när du implementerar A*. Varje `Node`-struktur har en medlemsfunktion `heuristic` som tar in en pekare till en annan nod och returnerar ett heuristikvärde som en `double`. Du kan anta att detta utgör en *giltig heuristik*, vilket betyder att avståndet till slutnoden aldrig överskattas (vilket är essentiellt för A*). Till exempel:

```
Node* node1 = graph.getNode("foo");
Node* node2 = graph.getNode("bar");
double h = node1->heuristic(node2); // compute A* heuristic between these nodes
```

Din algoritm för A*-sökning ska alltid returnera en väg med samma längd och kostnad som Dijkstras algoritm. Om du märker att algoritmerna hittar vägar med olika kostnader tyder det troligtvis på en bugg i din lösning. För labyrinter ska BFS, Dijkstras algoritm och A* hitta vägar av samma längd och kostnad.

Flera filer med förväntat utdata finns på kurshemsidan. Om du har implementerat alla vägsökningsalgoritmer korrekt ska DFS hitta någon giltig väg från start till mål; BFS ska få samma väglängder som i det förväntade utdatat på kurshemsidan. För Dijkstra och A* ska du få samma vägkostnader som i det förväntade utdatat. själva vägen behöver dock inte matcha exakt, inte heller de besökta positionerna, så länge som din väg är en korrekt väg.